Software Security using Software Dynamic Translation

Kevin Scott and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904

{kscott,jwd}@cs.virginia.edu

ABSTRACT

Software dynamic translation (SDT) is a technology that allows programs to be modified as they are running. Researchers have used SDT with good success to build a variety of useful software tools (e.g., binary translators, operating system simulators, low-overhead profilers, and dynamic optimizers). In this paper, we describe how SDT can be used to address the critical problem of providing software security. The paper shows how SDT can simply and effectively implement arbitrary user-specified software safety policies. Unlike static analysis techniques which typically process source code, SDT is applied to binary code. Consequently, SDT can handle untrusted binaries and unsecured libraries from any source. To demonstrate and validate that SDT provides additional security, we have implemented a software security API for Strata, our software dynamic translation infrastructure. The API, while simple, allows clients to implement powerful policies to prevent potential security violations. To illustrate the use of Strata and the security API, the paper provides implementations of several interesting and useful security policies.

1 INTRODUCTION

One of the most pressing problems for software developers is delivering secure software. Software security is vital given our dependence on software running and managing critical infrastructure. The cost of deploying insecure software is high. Estimates of the economic impact of the past few Internet viruses are in the billions: Code Red cost $1.2 billion; Melissa cleanup was $1 billion; and the Love Bug virus is estimated to have cost $8.7 billion [24]. Even if these estimates are off by a factor of ten, the total cost of dealing with just Internet viruses is extremely high.

Building software that works and that is delivered on time and within budget is difficult enough. Building software that is also secure adds yet another complication. This difficulty is evidenced by the large number of vulnerabilities discovered in our computer systems. For example, one security Web site lists over 100 known vulnerabilities to several commercial and open source operating systems [14]. Given its importance and the large costs when insecure software is deployed, it is not surprising that software security is being addressed on a number of fronts. Language designers are working to develop languages where security is a primary design goal [17]. Specialized compilers that produce code that thwart certain classes of exploits are being developed [7]. Static analysis techniques are being applied to catch vulnerabilities before software is deployed [27]. Research on embedding proofs in an application and verifying the proof before running the application is another promising approach [18, 20]. While progress is being made and the issues are being brought to the forefront [25], building and deploying secure software remains elusive.

In this paper, we describe a simple, yet powerful approach for helping provide software security. By software security, we mean the ability to enforce a policy that specifies how resources may be used [25]. A simple software security policy might be that a program may not read a particular file (e.g., `/etc/passwd` or `registry.dat`). More complicated policies may specify limits on the use of some resource. For example, we may wish to limit the rate at which an untrusted application can send packets over the Internet. This could be useful in slowing the spread of viruses or preventing denial of service attacks.

Our approach is to use software dynamic translation (SDT) to modify a running application so that a user-specified security policy is enforced. System managers and users write security policies using a simple software security API that was designed to work with Strata, our retargetable, extensible software dynamic translator. Strata modifies the running application so that the security policy is invoked when appropriate. Our approach has many advantageous features. Foremost, our approach can handle untrusted binaries. Source code is not required. This makes our approach particularly attractive for handling mobile code received from any source (including e-mail attachments). Second, because SDT deals with binary code, our approach is language and compiler independent. Third, our simple API can be used in conjunction with any programming language. This makes our approach accessible to a wide range of users. Finally because our approach

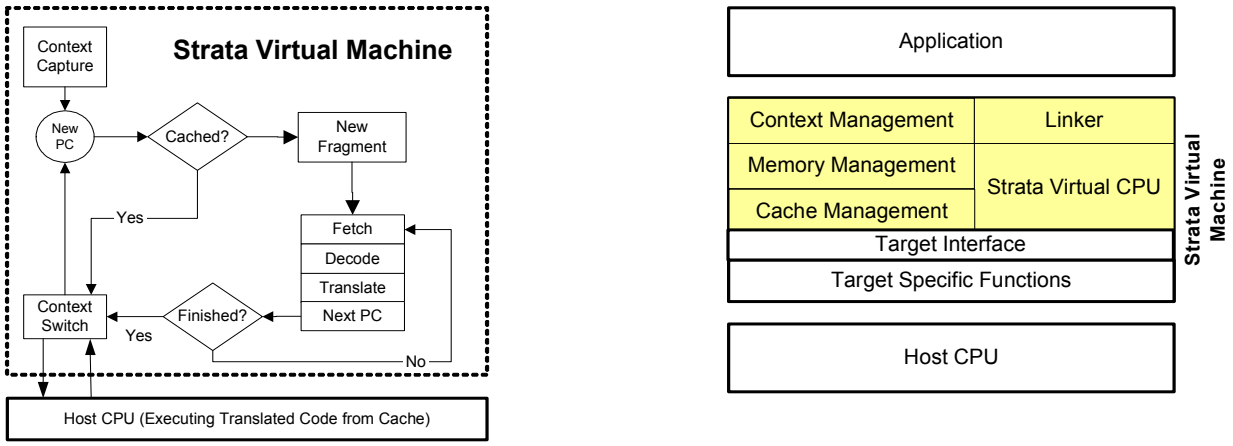

Figure 1: Strata Architecture

operates at run time, we can define and enforce security policies that are not possible with static analysis approaches.

This paper has the following organization. Section 2 briefly describes Strata, our SDT infrastructure. Section 3 describes Strata's security API and describes how Strata modifies a running application and enforces user-specified software security policies. In Section 4, we present several security policies to demonstrate both the power and simplicity of enforcing policies using Strata. Section 5 discusses the limitations of our approach. The final two sections discuss related work and provide a summary, respectively.

2 STRATA

Inspired by the success of Dynamo [3, 2], DAISY [10], UQDBT [23], and others [15, 26], we have constructed an infrastructure, called Strata, for exploring applications of software dynamic translation (SDT). To this end, Strata was designed with portability and extensibility in mind. Portability allows Strata to be moved to new machines easily. Thus researchers can explore architectural trade offs associated with the efficient implementation of SDT. Extensibility allows Strata to be used for a variety of different purposes; researchers can use Strata to build dynamic optimizers, dynamic binary translators, fast architecture emulators, etc. This paper describes how we extended Strata to build a system that enforces user-specified security policies.

Figure 1a illustrates how Strata controls and mediates the execution of an application binary. To run an application binary under Strata control, the binary is rewritten to replace the call to the program's entry point with a call to the Strata entry point. For C and C++ applications, the call to `main()` is replaced with a call to Strata's entry point. When this call is executed, Strata is dynamically linked with the application and Strata is invoked. Strata then saves the application state (context capture) and invokes the Strata component known as the fragment builder.

Strata's fragment builder takes the PC of the next instruction that the application binary needs to execute, and if the instruction at that PC has not been cached, the fragment builder begins to form a fragment. A fragment is a sequence of code in which branches may appear only at the end. Strata populates fragments by fetching, decoding, and translating application instructions until an end-of-fragment condition is met. When an end-of-fragment condition is met, Strata replaces the control transfer instruction with trampoline code, that when executed, returns control back to Strata. For the Strata-based security tools described in this paper, the end-of-fragment condition is met when a conditional or indirect control transfer instruction is fetched.

Once a fragment is fully formed, it is placed in the fragment cache and Strata performs a context switch that begins execution of the fragment. At the end of the fragment, the trampoline code causes a context switch back to Strata and the process of building the next fragment begins. As execution proceeds, the working set of the application materializes in the fragment cache and less and less work is done by Strata.

Figure 1b shows the high-level architecture of Strata. As the figure indicates, Strata is a software layer that separates the application from the host CPU and operating system. Strata's basic services implement a very simple dynamic translator that mediates execution of native application binaries with no visible changes to application semantics. The basic services include memory management, fragment cache management, application context management, a dynamic linker, and a fetch/decode/translate

engine. The basic services also include a few fundamental optimizations to ensure that the overhead of running an application under the control of Strata is minimal. These optimizations include partial inlining, fragment linking, hot path layout, and handling indirect branches efficiently.

Strata consists of 8000 lines of C code, roughly 30% of which is target-specific. It currently runs on both the SPARC and MIPS platforms (running Solaris and IRIX operating systems, respectively). A port for the IA-32 architecture for Linux and Windows is in development.

3 SOFTWARE SECURITY WITH SDT

SDT's ability to control and dynamically modify a running program provides the mechanism to enforce user-specified security policies on untrusted binaries. The basic idea is that the untrusted binary is enveloped by a Strata security layer. As the application is virtualized by Strata, code is dynamically inserted to enforce the user-specified security policy. Thus, access to host CPU and operating system resources and services is controlled by Strata (see Figure 2).

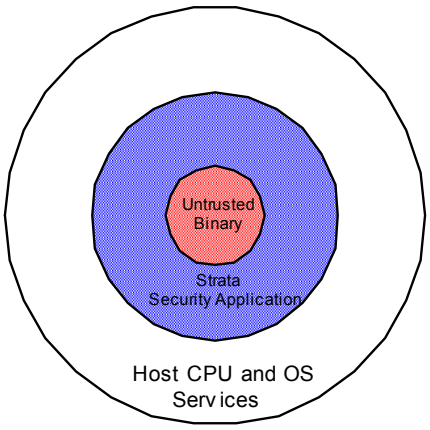

Figure 2: Strata

In this paper, we will use terms and phrases that are typically employed when discussing the Unix operating system (e.g., "becoming root", "exec'ing a shell", "performing a setuid(0)", etc.). The actions indicated by these terms have analogs in other major operating systems (e.g., Windows NT, Windows 2000, Window XP, VxWorks, and PSOSystem) and the approaches we describe would apply equally well to applications running on these systems.

A simple, but realistic example illustrates our approach. Suppose a user wishes to enforce a policy that prohibits untrusted applications from reading a file that the user normally has permission to read. Let's call this file `/etc/passwd` (`registry.dat`, `SAM`, or `system` might be equally good choices). Now assume the user receives an untrusted binary called `funny` and wishes to run it. The user invokes `funny` using the Strata security loader. The Strata security loader locates the entry point of the application and inserts a call to the Strata startup routine. When the loader begins execution of the application, the call to the Strata startup routine causes Strata to be dynamically loaded and invoked.

As Strata processes `funny`'s text segment and builds fragments to be executed, it locates open system calls and replaces them with code that invokes the security policy code. When the fragment code is executed, all open system calls are diverted to the policy code. It is the policy code's job to examine the arguments to the original open system call. If the untrusted binary is attempting to open `/etc/passwd`, an error message is issued and execution of the binary is terminated. If the file being opened is not `/etc/passwd`, the security policy code performs the open request and returns the result and execution continues normally (albeit under the control of Strata).

As we shall show in the following sections, our approach to using SDT to enforce security is simple, yet extremely powerful. Using Strata's security API users can implement powerful security policies that are not possible with static approaches.

3.1 Strata Security API

Our initial idea was to design a domain-specific language for expressing the security policies to enforce at runtime. These policies would be compiled into code that would be injected into the application by Strata. Our rationale was that a domain-specific language would more likely yield short, easy to write, and easy to read security policies.

Pursuing this approach, we completed a preliminary language design and wrote several sample security policies to evaluate our design. As a result of this evaluation, we realized several things. First, designing a new language is hard. The resulting language often looks a lot like other languages, but is just different enough to confuse people. Second, the real issue was not the language syntax, but providing the right set of primitives. Third, we would need to write a policy compiler that generated very high-quality code for the policies. Furthermore, the policy compiler would need to be retargetable so it could be used in conjunction with Strata on different platforms. Our conclusion was that it would be a lot of work to invent a new language and accompanying compilers, and we were not likely gain much by doing so.

Based on this preliminary design exercise, we decided that a better approach would be to develop a simple API and accompanying libraries that could be accessed using existing programming languages and their compilers. There were a couple of advantages of the API approach over the domain-specific language approach. Potential users would not have to learn a new language, and they could implement security policies using their favorite programming

language. A second benefit was that we would not have to build a special compiler for the policy language. We could use an existing mature compiler for the target source language and machine as long as it generated very high-quality code. High-quality code is important as the overhead of executing the dynamically injected policy code has to be kept as low as possible.

Once we abandoned the distraction of trying to design a new language and we focused on the real problem of defining the primitives needed to enforce software security our job became much easier. Recall our definition of software security. Software security is the ability to enforce a policy that specifies how resources may be used. What are the resources we need to manage? Most security vulnerabilities involve misuse of resources managed by the operating system.

For example, a hacker obtains root privileges by exec'ing a shell while the application is running in super-user mode. Both performing the exec and changing run-level privileges involve operating system calls (i.e, `exec` and `setuid`). Similarly, a hacker may read a file to gain information, or write false information into a file that may assist future attempts to compromise the system. Again, access to the file system is via operating system calls (i.e., `open`, `read`, and `write` in this example).

Thus to provide support for software security, Strata's security API should provide facilities for allowing users to write policies that specify how operating systems resources can be used. Fundamentally, Strata should provide a low-cost method for invoking a security policy when a relevant operating system call is made.

Our approach is to provide a simple, efficient facility that allows the user to specify which operating system calls to monitor and policy code to execute when the operating system call is invoked. Strata's security API consists of four functions. They are:

```
void init_syscall();
watch_syscall(unsigned num, void *callback);
void strata_policy_begin(unsigned num);
void strata_policy_end(unsigned num);
```

The first function is called on the initial entry to Strata. The implementation of this function will contain calls to the second API function `watch_syscall()`. Function `watch_syscall()` specifies an operating system call to watch (i.e., `num`) and the security policy to execute when that OS call is invoked (i.e., `callback`). The signature of `callback` should match the signature of the operating system call being watched. The final two API functions are used to bracket security policy code. The need for these two functions will be explained shortly when we describe how Strata dynamically injects the security policy code into the application.

To illustrate the implementation of Strata's security API, we show the Strata security policy for preventing an untrusted application from reading `/etc/passwd`. Following the style used on hacker websites to demonstrate exploitation of security vulnerabilities, we give a small demonstration program that exercises the policy. The demonstration code is given in Listing 1.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <strata.h>
4.  #include <sys/syscall.h>
5.  int myopen (const char *path, int oflag) {
6.      char absfilename[1024];
7.      int fd;
8.      strata_policy_begin(SYS_open);
9.      makepath_absolute(absfilename,path,1024);
10.     if (strcmp(absfilename,"/etc/passwd") == 0) {
11.         strata_fatal("Naughty, naughty!");
12.     }
13.     fd = syscall(SYS_open, path, oflag);
14.     strata_policy_end(SYS_open);
15.     return fd;
16. }
17. void init_syscall() {
18.     (*TI.watch_syscall)(SYS_open, myopen);
19. }
20.
21. int main(int argc, char *argv[]) {
22.     FILE *f;
23.     if (argc < 2 || (f = fopen(argv[1],"r")) == NULL) {
24.         fprintf(stderr,"Can't open file.\n");
25.         exit(1);
26.     }
27.     printf("File %s opened.\n",argv[1]);
28.     return 0;
29. }
```

Listing 1: Security policy for preventing a file from being opened.

Before explaining how Strata injects this policy into an untrusted binary, we review the code at a high level. Function `init_syscall()` at lines 17–20 specifies that `SYS_open` calls should be monitored and that when a `SYS_open` call is to be executed by the application, control is transferred to the policy routine `myopen()`.

Function `myopen()` (lines 10–25) implements the security policy. As mentioned previously, invocations of `strata_policy_begin()` and `strata_policy_end()` are used to bracket the policy code and their purpose will be explained shortly.

In function `myopen()`, the path to be opened is converted to an absolute pathname by calling the utility function `makepath_absolute()`. The path returned is compared to the string `/etc/passwd` and if it matches, an error message is issued and execution is terminated. If the file to be opened is not `/etc/passwd`, then the policy code performs the `SYS_open` system call and returns the result to the client application as if the actual system call was executed.

We now describe the mechanisms that Strata uses to selectively watch system calls and how it injects the policy code

into the running application. As shown in Figure 3, when an untrusted binary is to be executed, the Strata security loader modifies the application binary so that initial control is transferred to Strata's initialization routines. This routine dynamically loads and executes the `init_syscall()` function that sets up a table of system calls to watch and their corresponding callback functions.

After initialization is complete, Strata begins building the initial application fragment by fetching, decoding and translating instructions from the application text into the fragment cache (see Figure 3). For the Strata-based security applications discussed in this paper, the translate function examines the application code and locates operating systems calls.

For each operating system call site, Strata tries to determine if the operating system call is one to be monitored. In most cases, Strata can determine, at translation time, the operating system call to be invoked. In this case, if the OS call is one to be monitored, the code to invoke the operating system call is replaced with a call to the user-supplied policy code. If the call is not one to be monitored, no translation action need be taken and the operating system call code is copied unchanged to the fragment cache.

In some cases, the operating system call to be invoked cannot be determined at translation time. This can occur with indirect operating system calls (e.g., using syscall). In this case, Strata must generate and insert code that, when the fragment is executed, will test whether the OS call being invoked is one to be monitored and if so, call the appropriate user-supplied policy code; otherwise the OS call is executed.

In the case where the OS call to be invoked can be determined at fragment creation time, Strata treats policy code just like application code. As a result, calls to policy code can often be partially inlined improving the efficiency of the code. However, partially inlining policy code creates a complication. Consider the `myopen()` policy code in Listing 1. When this code is inlined, the `SYS_open` OS call will be generated. This OS call should not be replaced by a callback as it is the OS call to execute when the policy's conditions are satisfied. To avoid infinite recursion, policy code is bracketed using the API calls `strata_policy_begin()` and `strata_policy_end()`. Strata uses these "code markers" to suspend translation of operating system calls. Thus, we are assuming that the writer of policy code is not malicious.

There is one further complication. A malicious user with knowledge of how Strata operates may try to circumvent Strata by using calls to `strata_policy_begin()` and `strata_policy_end()` to bracket application code that attempts to violate the security policy. To prevent this avenue of attack, Strata only permits `strata_policy_begin()` and `strata_policy_end()` to execute from within security policy code.

4 SECURITY POLICIES

To illustrate both the power and simplicity of our approach, this section presents several other interesting security policies.

A common security exploit is to arrange to exec a shell while in root or super-user mode. This is most commonly done using a buffer overrun attack that corrupts the runtime stack. In an earlier paper we described how such an attack can be stopped using Strata's target-dependent interfaces [21]. Other types of attacks are possible [25]. However, they all rely on exec'ing a program (usually a shell) while in root or super-user mode. Using Strata's security API, it is very simple to write a policy that prohibits exec'ing a program when in super-user mode, yet allow exec's when not in super-user mode. Listing 2 contains the demonstration program.

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <unistd.h>
4.  #include <strata.h>
5.  #include <sys/syscall.h>
6.  static int curuid = -1;
7.  int mysetuid (int uid) {
8.      int retval;
9.      strata_policy_begin(SYS_setuid);
10.     curuid = syscall(SYS_setuid, uid);
11.     strata_policy_end(SYS_setuid);
12.     return retval;
13. }
14. int myexecve (const char *path, char *const argv[],
15.  char *const envp[]) {
16.     int retval;
17.     strata_syscallback_begin(SYS_execve);
18.     if (curuid == 0)
19.         strata_fatal("Naughty, naughty");
20.     retval = syscall(SYS_execve, path, argv, envp);
21.     strata_syscallback_end(SYS_execve);
22.     return retval;
23. }
24. void init_syscall() {
25.     (*TI.watch_syscall)(SYS_execve, myexecve);
26.     (*TI.watch_syscall)(SYS_setuid, mysetuid);
27. }
28. int main (int argc, char *argv[]) {
29.     FILE *f;
30.     char *args[2] = {"/bin/sh",0};
31.     setuid(0);
32.     execv("/bin/sh", args);
33.     return 0;
34. }
```

Listing 2: Security policy to prevent exec's while root.

To implement this security policy, we must monitor two system calls—`setuid` and `execve`. We must monitor `setuid` to keep track of the uid of the running application. This information is stored in the state variable `curuid`. In function `myexecve()`, if the program is running in root mode (i.e., the uid of the process is 0) exec's are disallowed, otherwise they are allowed.

The second security policy presented implements a policy that controls the rate that an application uses a resource.

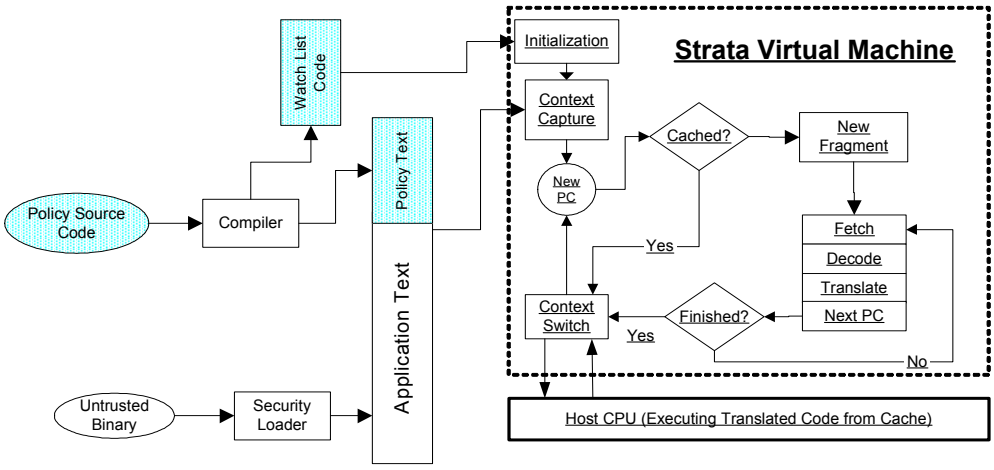

Figure 3: Injecting user-policy code into an untrusted binary using software dynamic translation.

In this example, we will limit the rate at which an application can transmit packets over a socket. This type of policy could be useful for thwarting denial of service attacks where zombie processes attempt to flood a server with packets. Listing 3 gives the code for the demonstration application.

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <sys/types.h>
4.  #include <sys/socket.h>
5.  #include <netinet/in.h>
6.  #include <netdb.h>
7.  #include <time.h>
8.  #include <string.h>
9.  #include <strata.h>
10. #include <sys/syscall.h>
11. #define RATE 10000
12. #define TOPRATE 10000000
13. #define DISCARD_PORT 9999
14. #define PAYLOAD_SIZE 1024
15. void xmit (const char *host, int nbytes);
16. static int socket_fd = -1;
17. /* Compute the delay necessary to maintain */
18. /* the desired rate */
19. int limiting_delay (double rate, time_t tbeg,
20.  time_t tend, int last_len, int len);
21. /* Callback for the so_socket call */
22. int my_so_socket (int a,int b,int c,char *d,int e) {
23.     int fd;
24.     strata_policy_begin(SYS_so_socket);
25.     /* Make the system call and */
26.     /* record the file descriptor */
27.     socket_fd = syscall(SYS_so_socket,a,b,c,d,e);
28.     strata_policy_end(SYS_so_socket);
29.     return fd;
30. }
31. /* Callback for the write system call */
32. int my_send (int s, const void *msg, size_t len,
33.  int flags) {
```

Listing 3: Security policy to limit the rate of transmission over a socket.

```
34.     int result;
35.     time_t now;
36.     static int last_len = 0;
37.     static time_t last_time = 0;
38.     strata_policy_begin(SYS_send);
39.     /* Only look at writes to socket_fd */
        /* and only rewrite HTTP headers */
40.     if (s == socket_fd) {
41.         now = time(NULL);
42.         sleep(limiting_delay(RATE,last_time, now
43.          len,last_len));
44.         last_len = len;
45.         last_time = now;
46.     }
47.     result = syscall(SYS_send,s,msg,len,flags);
48.     strata_policy_end(SYS_send);
49.     return result;
50. }
51. void init_syscall() {
52.     (*TI.watch_syscall)(SYS_so_socket,my_so_socket);
53.     (*TI.watch_syscall)(SYS_send,my_send);
54. }
55. main(int argc, char *argv[]) {
56.     if (argc == 3)
57.         xmit(argv[1],atoi(argv[2]));
58.     else
59.         fprintf(stderr,
60.          "Usage: %s host nbytes\n",argv[0]);
61. }
62. /* Transmit nbytes to discard port (9) on host */
63. void xmit (const char *host, int nbytes) {
64.     int sd, bytes_sent;
65.     struct sockaddr_in sin;
66.     struct sockaddr_in pin;
67.     struct hostent *hp;
68.     char *payload[PAYLOAD_SIZE];
69.     time_t begin, elapsed;
70.     double rate;
71.     /* go find out about the desired host machine */
72.     if ((hp = gethostbyname(host)) == 0) {
73.         perror("gethostbyname");
```

Listing 3: (Continued) Security policy to limit the rate of transmission over a socket.

```
74.        exit(1);
75.    }
76.    /* fill in the socket structure with host info */
77.    memset(&pin, 0, sizeof(pin));
78.    pin.sin_family = AF_INET;
79.    pin.sin_addr.s_addr = ((struct in_addr *)
80.         (hp->h_addr))->s_addr;
81.    pin.sin_port = htons(DISCARD_PORT);
82.    /* grab an Internet domain socket */
83.    if ((sd = socket(AF_INET,SOCK_STREAM, 0)) == -1) {
84.       perror("socket");
85.       exit(1);
86.       }
87.    /* connect to PORT on HOST */
88.    if (connect(sd, (struct sockaddr *) &pin,
89.         sizeof(pin)) == -1) {
90.       perror("connect");
91.       exit(1);
92.    }
93.    begin = time(0);
94.    bytes_sent = 0;
95.    while(bytes_sent < nbytes) {
96.       /* send a message to the server PORT */
97.       /* on machine HOST */
98.       if (send(sd,payload,sizeof(payload),0) == -1) {
99.          perror("send");
100.         exit(1);
101.      }
102.      bytes_sent += sizeof(payload);
103.      printf(".");
104.      fflush(stdout);
105.   }
106.   elapsed = time(0) - begin;
107.   rate = bytes_sent / elapsed;
108.   printf("\nRate = %8.3f bytes per second.\n",rate);
109.   close(sd);
110.}
```

Listing 3: (Continued) Security policy to limit the rate of transmission over a socket.

To implement this policy, `SYS_so_socket` and `SYS_send` system calls must be monitored. Callbacks `SYS_so_socket (my_so_socket)` and `SYS_send (and my_send)` are established (lines 49–52). The policy code for monitoring the socket call simply records the file descriptor for the socket. The recorded file descriptor will be used by `my_send()` to limit the rate only on this connection. In function `my_send()`, if the transmission is to the monitored connection (i.e., `socket_fd`), then a delay is introduced if necessary (see line 42 of Listing 3).

Listing 4 contains our third and final Strata security demonstration program. The security policy prevents cookies from being transmitted to web servers. In this example, the two system calls to be monitored are `SYS_so_socket` and `SYS_write`. Like the previous example, the callback `my_so_socket()` simply remembers the socket being opened. In callback `my_write()`, writes to the socket are detected and the buffer is preprocessed by `remove_cookies()` before writing it (Listing 4 lines 30–34).

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #include <strata.h>
4.  #include <sys/syscall.h>
5.  #include "snarf.h"
6.
7.  static int socket_fd = -1;
8.
9.  /* Copy src buffer to dst removing cookies*/
10. int remove_cookies(char *dst, const void *src,
11.  int size);
12. /* Callback for the so_socket system call. */
13. int my_so_socket (int a, int b, int c, char *d,
14.  int e) {
15.    int fd;
16.    strata_policy_begin(SYS_so_socket);
17.    /* Make the system call and record the */
18.    /* file descriptor */
19.    socket_fd = syscall(SYS_so_socket,a,b,c,d,e);
20.    strata_policy_end(SYS_so_socket);
21.    return socket_fd;
22. }
23. /* Callback for the write system call */
24. int my_write (int fd, void *buf, int size) {
25.    char new_buf[1024];
26.    int s, new_size;
27.    strata_policy_begin(SYS_write);
28.    /* Only look at writes to socket_fd
29.    /* and only rewrite HTTP headers. */
30.    if (fd == socket_fd &&
31.     (new_size = remove_cookies(new_buf,buf,size)))
32.       s = syscall(SYS_write,fd,new_buf,new_size);
33.    else
34.       s = syscall(SYS_write,fd,buf,size);
35.    strata_policy_end(SYS_write);
36.    return s;
37. }
38. void init_syscall() {
39.    (*TI.watch_syscall)(SYS_so_socket,my_so_socket);
40.    (*TI.watch_syscall)(SYS_write,my_write);
41. }
42. int main(int argc, char *argv[]) {
43.    snarf_main(argc, argv);
44. }
```

Listing 4: Security policy to remove cookies.

5 DISCUSSION

As the previous section has shown, writing powerful software security policies using Strata's security API is simple. While it is somewhat ironic that we wrote our sample security policies using C which is a cause of many of the security vulnerabilities, we did so to make the techniques and policies accessible to the largest audience. The policies presented could have been written in any language. The only requirements are that bindings of Strata's security API must implementable in the new target language and that a compiler for the target language must be available that emits object code. Providing a new language implementation of Strata's security API is simple since the API consists of four simple functions.

Airtight security is extremely difficult to provide. Code claimed to be secure is often compromised. We do not

claim that Strata's security API can foil all possible attacks. However, it does provide some additional level of protection. Furthermore, it allows easy implementation of policies that are beyond static approaches. Indeed, all three policies shown in Section 4 could not be enforced using static approaches as they require state information that can only be determined dynamically (e.g., the user id when an exec is performed).

Our current implementation does have some limitations on the types of security policies that can be enforced. For instance, Strata does not currently handle multi-threaded code. There are clever attacks that exploit timing vulnerabilities in threaded code. We are currently extending Strata to handle threaded code and we plan to investigate developing policies to prevent this type of attack. Nonetheless, we feel that our approach provides a useful and complementary approach to helping provide software security.

Another important issue to consider is the overhead of using SDT for secure software. Noticeable or significantly high overheads will limit the applicability of using SDT to address software security. In a previous paper, we discussed techniques for reducing overhead and showed that SDT was competitive with previously developed techniques for preventing certain classes of security breaches [22]. Currently, the slowdown of running an application under Strata varies but can be as much as 1.32X. For many types of applications, 30 percent overhead is acceptable. Examples include executing an e-mail attachment that includes a self-extracting archive, opening a foreign document that contains malacious macros that destroy valuable information, and many set-user-id programs that perform simple administrative functions. For these situations, a slowdown of 20 to 30 percent would not be noticeable to the user.

For some applications such as web servers, web browsers, and databases, an overhead of 30 percent might not be acceptable. Indeed for these types of applications any overhead is unlikely to be acceptable. Fortunately, previous research on dynamic optimization has shown that it is possible to achieve substantial speedups in long running applications [4, 2, 15]. Thus we believe that by combining a dynamic security checker with a dynamic optimizer CPU-intensive applications can be run securely without overhead. To this end, we are working to incorporate additional optimizations within Strata's framework. If successful, our approach to software security would be pplicable to an even wider range of applications.

6 RELATED WORK

Enforcing security policies through software has been the focus of much recent research. Approaches vary from static, source code analysis, to dynamic approaches such as sandboxing and execution monitoring. The different approaches vary in flexibility and their ability, or lack thereof, to enforce certain types of security policies.

Static, source-code analysis can be used to enforce a security policy by identifying, at compile time, those programs that could violate the policy. The obvious advantages of this approach are early identification of security policy violations, and little or no run-time overhead. Early identification allows potential security problems to be corrected before applications are "shipped." A number of static security policy enforcement mechanisms have been proposed in the research literature, and some have even found their way into wide use.

Perhaps the most familiar of the static security policy enforcment mechanisms is type checking [6]. In compilers for typed languages, the type checker statically rejects those programs that could cause execution errors through misuse of resources, e.g., improperly referencing a memory location. The security policies enforceable through type checking are limited by the language's type system. For example, it may be impossible to prevent an application from consuming too many CPU cycles through type checking alone. However, type checking in strongly typed languages can enforce a number of important and useful security policies, e.g., "buffer overflows are not permitted."

Another class of static security policy enforcement mechanisms performs source code analysis, but does not rely exclusively on the programming language's type system. The annotation-assisted static checking system of Evans and Larochelle uses programmer supplied annotations to detect possible buffer overflow vulnerabilities in C programs [16]. Their system is specialized to one specific, albeit extremely important, security policy for one specific language. Further their system requires the programmer to supply special annotations in order to improve accuracy.

Proof carrying code is a bit more flexible, in the sense that a wider variety of security policies can be enforced [18, 20, 19, 1]. The proof carrying code producer, often in conjunction with the language type system, generates a proof that a particular program adheres to a consumer's security policy. If the consumer is able to verify the proof, it is assured that the program will not violate the security policy. Even though proof carrying code is rather flexible, there are limits to its abilities. Proof carrying code relies either on programmer annotations or type information from the underlying programming language in order to generate proofs. Further, proof carrying code cannot be used to enforce some security policies, e.g., those that involve temporal assertions.

Recently, many static source code analysis tools have been proposed, each of which is designed to detect programming mistakes. In some sense, execution errors due to programming mistakes may be viewed as violations of a security policy. Meta-level compilation has been successfully used to statically detect errors such as "pointer dereferenced before checked for NULL" [11]. The Vault system allows the programmer to specify interactions between programming modules [9]; the compiler is then able to statically verify that the program adheres to the specification. The RATS

source code auditing tool detects bad programming practices that may lead to a variety of security violations such as buffer overflows and illegal use of APIs [25].

Even though static approaches to security policy enforcement are popular, they have their limitations. As we have noted above, these limitations include restrictions on the types of security policies that are enforceable, and language dependences. Dynamic security policy enforcement tools address some of these concerns.

There are a variety of tools which dynamically prevent buffer overflow attacks. The StackGuard system is a special C compiler that generates special code to dynamically detect and prevent the occurrence of most stack buffer overflows, i.e., a "stack-smashing" attack [8]. The libverify tool uses a combination of late program modification and techniques borrowed from StackGuard to prevent buffer overflows, but without requiring a special compiler or access to source code [5]. While both of these tools are very useful, they are restricted to one specific security policy, one programming language, and in the case of StackGuard, require a special compiler.

Perhaps the most popular of the dynamic security policy enforcement techniques is software fault isolation, or sandboxing. Software fault isolation uses SDT to allow fast, safe RPC between isolated software modules [28]. In essence, each untrusted module is sandboxed, as if in it's own address space. This approach reduces the likelihood that an untrusted module can corrupt data in other modules. It is easy to see this as a security enforcement mechanism if we accept that isolation is a valid security policy. Sandboxing is also employed in the Java Virtual Machine to enforce an even wider variety of security policies [17]. For example, applets downloaded and run on a Web browser's JVM may be restricted from accessing data stored on the local disk.

The last of dynamic security policy enforcement approaches is execution monitoring. The Janus system monitors system calls executed by a program in order to determine whether or not a security policy has been violated [13]. Janus uses the operating system ptrace facility to register callbacks to policy enforcement code. Not all operating systems have such a facility, precluding Janus's use on those platforms. Janus also refrains from monitoring frequently executed system calls, e.g., write(), in order to keep overhead low. The SASI system, like Janus, also performs execution monitoring [12]. Rather than relying on an operating system facility, or restricting itself to monitoring only system calls, SASI inserts monitoring code required by the security policy directly into the program binary before execution. Consequently SASI cannot enforce the security policy on self-modifying code, or dynamically generated code.

Of all related systems, Janus appears most similar to the software security approach advocated by this paper. In both Janus and our Strata applications, security enforcement is accomplished by monitoring system calls. This is where the similarities end. We have already seen that the two systems monitor system calls using different mechanisms, and have pointed out that Janus is unable to efficiently handle certain types of system calls. However, the most important difference between Strata and Janus is that Janus cannot effect the actual execution behavior of the program, outside of aborting execution or providing notification. Strata on the other hand can arbitrarily alter system call semantics to meet the security policy enforcement needs of the user. We have seen this already in the snarf example, where Strata rewrites HTTP requests to remove cookies. In this sense Strata is a good deal more flexible, and powerful than Janus.

7 SUMMARY

One of the most pressing and important problems faced by software developers today is how to deliver secure software. Addressing this problem effectively is vital given the interconnectedness of the global computing infrastructure. Failure to provide secure software can cost billions. In this paper, we have presented a powerful approach to software security. Using a simple security API, users and administrators can write powerful, arbitrary security policies. Using software dynamic translation, these policies are injected into a running program. The SDT approach is attractive for several reasons. Foremost, the approach can handle untrusted binaries. Source code is not required. This makes our approach particularly attractive for handling mobile code received from any source (including e-mail attachments). Second, because SDT deals with binary code, our approach is language and compiler independent. Third, our simple API can be used in conjunction with any programming language. This makes our approach accessible to a wide range of users. Finally because our approach operates at run time, we can define and enforce security policies that are not possible with static analysis approaches.

8 REFERENCES

1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instuctions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 243–253, N.Y., January 19–21 2000. ACM Press.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical report, Hewlett-Packard Laboratories, June 1999.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[4] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings ACM SIGPLAN'2000 Conf. on Programming Languages Design and Implementation*, pages 1–12, June 2000.

[5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack-smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 251–262, Berkeley, CA, June 18–23 2000. USENIX Ass.

[6] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, FL, 1997.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 63–78, Berkeley, January 26–29 1998. Usenix Association.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, , and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 1998 USENIX Security Symposium*, 1998.

[9] Robert DeLine and Manuel Fähndrich. Enforcing High-Level protocols in Low-Level software. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 59–69, N.Y., June 20–22 2001. ACMPress.

[10] Kemal Ebcioglu and Erik Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[11] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 October 2000.

[12] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999. ACM SIGSAC, ACM Press.

[13] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[14] HoobieNet. Security exploits. http://www.hoobie.net/security/exploits/, 2001.

[15] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[16] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, 2001.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, USA, 1997.

[18] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Safe, Untrusted Agents using Proof-Carrying Code*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, Berlin Germany, 1998.

[19] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[20] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Canada, 17–19 June 1998.

[21] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the 2001 IEEE Workshop on Binary Translation (WBT)*, 2001.

[22] Kevin Scott, Jack Davidson, and Kevin Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, University of Virginia, July 2001.

[23] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo '00*, 2000.

[24] USA Today. The cost of code red: $1.2 billion. http://www.usatoday.com/life/cyber/tech/2001-08-01-code-red-costs.htm, 2001.

[25] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, Reading, USA, 2001.

[26] Michael Voss and Rudolf Eigenmann. A framework for remote dynamic program optimization. In *Proceedings of the ACM Workshop on Dynamic Optimization Dynamo '00*, 2000.

[27] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, February 2000. Internet Society.

[28] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, 1993.